Assignment 6 Solutions

Question 1

- TSP is NP-complete, which means that there is no-known polynomial-time algorithm. It is possible that one exists, since it is unknown whether or not NP=P.
- $n \times n$ chess is in the class EXP, and it is know to *not* be in the class P. This means that there **does not exist** a poly-time algorithm for this problem, but there does exist an exponential-time algorithm.
- The halting problem is proven to be undecidable. This means that there **does not exist** an algorithm that correctly solves this problem for all inputs. Therefore there is certainly not a poly-time algorithm.
- Vertex Cover is NP-complete, which means that there is no-known polynomial-time algorithm. It is possible that one exists, since it is unknown whether or not NP=P.
- Integer Facorization has not been shown to be NP-complete. The exact difficulty is not known. There is currently **no known** poly-time algorithm that solves all inputs to this problem, but it has not been shown that one doesn't exist.
- This problem is referred to as *Bin Packing*, which is an NP complete problem. There is no-known polynomial-time algorithm. It is possible that one exists, since it is unknown whether or not NP=P.

Question 2

1. $T(n) = (\log n)^6 \leq n^6$, and so in class P and in class EXP.
2. $T(n) = \log(n^6) \leq 6 \log n \leq 6n$, and so in class P and EXP
3. $T(n) = (6n)^6 = 6^6 n^6$ and so in class P and EXP
4. $T(n) = n + 1000$
5. $T(n) = n^n$. We can show that $n^n \leq 2^{n^2}$, which shows $T(n)$ is in EXP.

   $$\log(n^n) = n \log n \leq n^2$$

   which proves that $n^n \leq 2^{n^2}$ by taking 2^* to each side. Note that n^n is NOT in P since it cannot be upper bounded by a polynomial
6. $T(n) = 3^n + n^6 \leq 2 \cdot 3^n$ for large n. Note that $3^n \leq 4^n = 2^{2n}$, and so the problem is in EXP but NOT in P
7. $T(n) = 3^{n^2+6} = 3^6 3^{n^2} \leq 3^6 4^{n^2} = 3^6 2^{2n^2}$, and so the problem is in EXP but NOT in P.

Question 3

(a) We can model this problem as a graph where each of the m issues is a vertex, and each participant is an edge. If participant p is interested in issues u and v, then participant p becomes the edge between corresponding vertices u and v. The problem of selecting at most k issues so that each person is interested in at least one of the issues is then equivalent to **Vertex Cover**, which is NP complete. Therefore no known poly-time algorithm exists.

(b) The problem has a **polynomial-time** algorithm. We can enumerate all possible sets of three vertices, of which there are $\binom{n}{3} = O(n^3)$. For each set of 3, we can check by brute-force if those vertices form a cycle.

(c) If $k = n$, then the problem is equivalent to Hamilton Cycle, which is **NP-complete**. Therefore no known poly-time algorithm exists.

(d) The problem has a **poly-time** algorithm. We can run BFS from vertex s (or DFS). All visited vertices will be marked as *visited*. Then we loop through all vertices to ensure they are all visited. If not, we return FALSE.

(e) The problem has a **poly-time** algorithm. We can run the SCC algorithm, and in class we saw that each strongly-connected component is stored as a DFS-tree. Therefore in the second step of the SCC algorithm, we simply ensure that all vertices are visited after the first DFS-tree is created. If not, we return FALSE.

(f) The problem has a **poly-time** algorithm. We can run BFS from vertex s, which finds the shortest distance from vertex s to all other vertices in G. Recall that this distance is stored in $v.d$. Afterwards, we loop through all vertices and verify that $v.d \leq k$. If not, we return FALSE.

(g) The problem has a **poly-time** algorithm. We can run the Find-Cycle algorithm from class (version for directed graphs), which returns whether or not there is a cycle. If not, then we can conclue G is a DAG. Otherwise we return false.

(h) The problem has a **poly-time** algorithm. We can run Dijkstra's algorithm from vertex s, which finds the shortest weighted distance from vertex s to all other vertices in G. This distance is stored in $v.d$. After completion, we check if $t.d < k$, and if not, we return FALSE.

(j) This is equivalent to the Travelling Salesman *Path* problem, which like TSP is also NP-complete. Therefore there is no known poly-time algorithm.

Question 4

Show that the **PoliMeet** problem is NP-complete.

Step 1:. We can verify a proposed solution to the problem is polynomial time. Given a set S of proposed issues, we do the following

- Loop through the set S to ensure it has size at most k

-Loop through each participant, and loop through each of their list of issues, checking at least one of those issues is in S.

The above loops run in polynomial time.

Step 2: Reduction from VERTEX COVER. Given an input graph G to vertex cover, we translate it into input to the PoliMeet problem as follows: Each vertex becomes an issue, and each edge becomes a participant. For each edge that connects vertices u and v, that corresponds to a participant who is interested in issues u and v. Note that in the input to the PoliMeet problem, each person only has 2 issues on their list. Now look for at most k issues that can be selected so that each participant is interested in at least one of them.

If the input to Vertex Cover is YES, then the corresponding input to PoliMeet is YES: If the graph has a set of at most k vertices that cover the edges, then exactly those k issues will cover the participants, since each edge that is covered in G is a participant which is covered in the Poli-Meet problem.

If the input to Vertex Cover is NO, then the corresponding input to PoliMeet is NO: If G has NO vertex cover of size k, then no set of k vertices cover all the edges. This means it is impossible to have k issues that satisfy the Poli-meet problem, since if so, those issues would be a vertex cover of size k

Question 5

Show that the **TravelFromS** problem is NP-complete.

Step 1:. We can verify a proposed solution to the problem is polynomial time. Given a path P through the input graph:

- Verify that the start vertex is S
- Loop through adjacent pairs of vertices in P and ensure that there is an edge in G for each adjacent pair in the path
- Loop through the vertices of G to ensure that each vertex is visited in P

The above loops run in polynomial time.

Step 2: Reduction from HAMILTON PATH. Given an input graph G to HamPath, we translate it into input to the TravelFromS problem as follows:

The input graph G is copied as input to the TravelFromS problem. A new vertex S is added, and the new vertex is connect to *every other vertex in* G. This new vertex is set as the source input vertex to the TravelFromS problem.

If the input to HamPath is YES, then the corresponding input to TravelFromS is YES: If the original graph G has a Hamilton path, then a path exists that visits all vertices. Suppose this path starts at vertex x. Then the new input to TravelFromS has a path from S, since the path corresponds to S-¿x-¿remaining vertices.

If the input to HamPath is NO, then the corresponding input to TravelFromS is NO: If G has NO Hamilton Path there is no path that starts at any vertex that visits all other vertices. Therefore in the input to TravelFromS, there is no way to start at vertex S and then travel to some other vertex of G to complete a path that visits all vertices.